TIDE

Telephone- **I**nterviewing ready **D**ata **E**ntry Program

Version 1.12

Outline of Documentation

I. GENERAL INTRODUCTION
II. THE CODEBOOK—BASICS
- A. Question and Response Categories
- B. Skip Patterns
- C. Pre-Entry Codes

III. OPERATION
- A. Files in and Out
- B. Routine Operation
- C. Real Time Changes and Fixes

IV. LISTS
- A. Sorting
- B. Multiple Punch

V. USE OF OPERATORS
- A. General
- B. Operators
- C. Stack Use

VI. FILES AND LOOPS
- A. Types of Files
- B. Grouped Files
- C. Dyadic Files
- D. Loops

VII. USE OF CONDITIONS
VIII. HAVING TIDE SET UP THE CODEBOOK FOR YOU
- A. Basics: Question Numbers and Skips
- B. Response Categories and Mnemonics
- C. Formatting Assumptions

IX. READING IN THE DATA
- A. General Issues
- B. Creating Variable Names

X. SYNTAX CHART

Documentation for **T**elephone **I**nterview-ready **D**ata **E**ntry (TIDE)

I. GENERAL INTRODUCTION

TIDE can be used as a simple CATI system, or to facilitate coding of already gathered data. There are three parts to using TIDE: setting up a codebook, using the program, and reading in the finished data. Almost all the work goes into creating the codebook, a relatively complex set of instructions which allows TIDE not only to prompt for questions, but to execute skip patterns based on what a respondent answers, on previously existing data about that respondent or a group to which she belongs, or on other considerations brought in by the analyst. Because to understand the codebook one must understand TIDE's structure, this documentation first introduces a simple form of the codebook, and then discusses some of the more important complications, before explaining how one would actually go about setting up a codebook. Once the codebook is done, TIDE is extremely easy to operate; further, TIDE saves the data in a cumulative file and simultaneously creates the SPSS syntax to read in this file.

TIDE has been made more user friendly as it progresses; consequently, the codebook tends to start with more ungainly ways of doing things and then introduce more abstract and elegant ways. This may prove helpful, for the reader will get an understanding of the basics of TIDE's approach that would otherwise be obscured, but it is worth skimming over the sections on letting TIDE set up a codebook for you and on MNEMEs before doing much work.

In this documentation, the following conventions are used: "<" and ">" brackets enclose a field the precise value of which you must supply. A descriptive label for the field is put between the brackets, for example, <variable number> indicates that you must supply some variable number. You would not write the < > brackets. Words that are used in a codebook are generally capitalized—if the TIDE word is shorter than the English word—for example, "operators" are referred to in the codebook as "OPER"—only the letters used by TIDE are capitalized. Hence this documentation will write "OPERator." Examples of actual codebook syntax are placed in courier font. A syntax chart for some command will be placed in the main font of this document. Hence the courier font will show one particular example of the more general class of command or syntax.

💣 Where you see this symbol, you know that you are being warned about a potential act that can cause TIDE to crash; be extremely attentive.

! Where you see this symbol, there is an important warning that you should pay attention to, in order to avoid a problem, but the problem is usually non-fatal.

Note indicates a less crucial, but important, piece of information brought to your attention.

➢ Where you see this symbol, there is a little tip that can save you some time or help you out in a jam.

II. THE CODEBOOK—BASICS

A. Questions and Response Categories

We will jump right in and start to examine the basics of codebook structure, examining a "finished" codebook. You would not actually write the codebook in this form; if you did, it would be very tortuous to make changes like inserting or deleting a question. Instead, you set things up in a more flexible way, and TIDE will edit it. But we'll worry about that later (see section VII, HAVING TIDE SET UP A CODEBOOK FOR YOU.) Let's start with the basic question, what is a codebook?

The codebook is a set of instructions that takes TIDE through a questionnaire. It is a text file, in the same directory as TIDE. The codebook file needs to have the following extension: .cbk. In its most basic form, for each question, the codebook contains has the text of the question and the possible response categories. Of course, to make things easy on TIDE, there are other things there, and there can be complications in the structure. Perhaps the best way is to look at a sample of an actual TIDE codebook.

```
{{Q0005}}I am sure that I would rather have children than not have children.
{END6}
{1}Strongly Agree
{2}Agree
{3}No Opinion
{4}Disagree
{5}Strongly Disagree
{6}Mixed
{{NEXT}}
{{Q0006}}Time spent with the group on a typical day.
{END4}
{L}0
{H}24
{{NEXT}}
! Here we switch to the next section
{{LINE}}
{{TALK}}Now I'd like to ask you a few questions about your views on
the world and what this nation should be doing.
{{NEXT}}
{{PAUSE}}
{{Q0007}}Looking at the state of the world today, I don't think people
get what they deserve.
{END1}
{1}Strongly Agree
{2}Agree
{3}No Opinion
{4}Disagree
{5}Strongly Disagree
{6}Mixed
{7}Other
{{FIN}}
```

First of all, you've probably noticed some things enclosed in curly brackets. These are things that TIDE considers very important. The first thing is the question number. This needs to start

{{Q and then have a FOUR DIGIT number. So {{Q0001}} is ok, {{Q1}} is not. But (see section VII) TIDE will take care of all this for you. There are two ways to refer to a question number; **absolute** (which is a four digit integer) and **relative** which is relative to the current position. The finished codebook only has absolute references, but as you'll see, you may set up a codebook using relative references.

Why the curley brackets? Anything in these curley brackets is a special codebook "structure" word. Those you see here are as follows:
{{Q<#>}}--indicates a question number
{END<#>}--indicates the end of question wording and what the response category type will be
 #--is a "response category"
{{NEXT}}—indicates we are done with a question or with a section of text
{{TALK}}—indicates that what follows is text to be spoken, and not a question.
{{LINE}}—indicates that TIDE should print a blank line
{{PAUSE}}—tells TIDE to stop printing and wait for the operator to press ENTER before
 continuing. Used to keep text from scrolling off the screen prematurely.
{{FIN}}—indicates that TIDE should stop.

Other structure words you have not yet seen are:
{{SKIP}}
{{CYCLE}}
{{MARK}}
{{POP}}

All the words begin with double curley brackets except for {END}; this is because {END} is considered "attached" to the {{Q}} structure word, and doesn't have as much of a quality of "beginning" something as the other wrods.

The TALK command would be used for a CATI implementation. As you can see, you simply precede with with {{TALK}} in double brackets. (TALK is used because TEXT is too close to NEXT.) Such comments can be inserted only between questions. Remember, TALK has to end with a {{NEXT}} statement, just as if it were the end of a question. Note that you cannot SKIP into a TALK section, but there is nothing to prevent you from putting "talk" type text into a question. Judicious use of TALK and in-question text can combine with SKIPs and IF structures to do pretty much whatever you want.

Third, you will see some lines beginning with !. These are comment lines; anything on such a line is ignored by TIDE, as long as it comes at an appropriate time, which is not in the middle of text (either talk or question), nor in the middle of response categories. By default, TIDE does not write comment lines to the log file. If you want such lines written (for example, the trick to map operator mnemonics to numbers discussed in VIII B), start each line with !!. **Note**: blank lines can also come at this opportune point, and they will be ignored. But they are not printed to the screen. To do this, use {{LINE}}.

What follows on the line after {{Q0005}} is the text of the question. If this flows onto the next line, that is fine, it should just start right in. BUT there must be a carriage return at the end of every line, and there cannot be more than 90 characters per line. (I'd suggest truncating lines at around 70 just to be safe, since there are various reasons why line length gets increased. See section VIII.) We see a multi-line question in Q7. TIDE just keeps on reading the question until it gets to {END#}.

When it comes to the category labels, however, TIDE assumes that each will occupy only one line. If you need to have a category label flow onto a second line, begin the second line with a '+' character. Below is a valid example.

```
{{Q0092}}Which of the following statements comes closest to expressing your
present
belief about God?
{END1}
{1}I don't believe in God
{2}I don't believe or disbelieve in God.  I don't think it's possible for us
+ to know anything about the existence of God
{3}I'm uncertain but lean toward not believing in God
{4}I'm uncertain but lean toward believing in God
{5}I definitely believe in God
{6}I'm uncomfortable about the word God, but I believe in something more or
+ beyond the world as we see it.
{{NEXT}}
```

Response Category Types (END)

Now what is this number following END? It is a key to TIDE telling it what kind of responses can be allowed. Note that type 15 is extremely general and useful (a late introduction, but will probably be preferred to most others). The key is as follows:

(0) This indicates that the question number comes from an internal process, such as an operator. This would be used if you want to save a random number that was used to determine which treatment some person received.

(1) there are a finite number of **one character** categories from which to choose (a "choice" question), a blank (no answer) is acceptable, and there is no room for extended "other" responses (compare to (2,6) below); NOTE: This **must** be a single character. This holds for options 2 and 6 below.

(2) there are a finite number of **one character** categories from which to choose (a "choice" question), there is no room for extended "other" responses (compare to (6) below), but a blank (no answer) is **not** acceptable. TIDE will not continue until this is filled in. This might be a piece of vital information that you cannot do without.

(3) The question has an integer range (a "range" question). Integers may in principle go from –9999 to +9999, but a range will probably be limited;

(4) The question has an integer range, but blanks responses are **not** permitted;

(5) The answer is expected to come in free form text (an "open" question);

(6) there are a finite number of categories from which to choose (a "choice" question), a blank (no answer) is acceptable, and there **is** room for extended "other" responses. TIDE automatically appends a possible response (see section II) which, if chosen, puts you in the same place you'd be if it were a totally open ended question. In the example, Q5 is of this form. If the person said "other" to

Q5, we'd get her actual words. But if the person said "other" to Q7, we'd just score it as other and never know what it was. It is expected that this format would be used for questions that are being tested or where the closed choices are expected to be supplemented with post-facto categories. **For such an other response**, TIDE always looks for '&' as your response category. It will print this to the screen automatically; you do not need to enter it into a codebook if you specify END6.

(7) This is the same as category 1, but only numeric data are allowed, i.e. the characters 0,1,2,3,4,5,6,7,8,9. Note that these need not be contiguous integers.

(8) This is the same as category 2, but only numeric data are allowed, i.e. the characters 0,1,2,3,4,5,6,7,8,9. Note that these need not be contiguous integers.

(9) This is the same as category 6, but only numeric data are allowed, i.e. the characters 0,1,2,3,4,5,6,7,8,9. Note that these need not be contiguous integers.

(10) A single ASCII name is written in; TIDE will search the NAMES file (see below, section V) for this name and write the corresponding ID number as a five digit integer.

(11) A series of ASCII names is written in of unknown length. TIDE will search the NAMES file, and then write to a dyadic data file a set of records that has only a single valid variable written to it, which is 1 if this person was named and zero otherwise. (NOT FINISHED)

(12) A LIST is to be sorted with TIDE assisting in the sorting process.

(13) A LIST allowing for multiple punches. This is the way TIDE assumes you will handle the possibility of choosing more than one. You first define a LIST of alternatives, then define the question, give and END13, and TIDE will expand it into a set of dichotomies. See section IV B.

(14) A single text line will be input. Response cannot flow onto more than one line.

(15) Less than 100 un-numbered response categories that TIDE will number (see below). An extended other response, however, is not currently permitted.

Depending on the type of response expected, what comes below the question wording is different. If it is an open question, nothing comes here. If it is a choice question, we have each of the response categories listed. The entry code—what the operator types in—is a **single** character. It can be a number or a letter for categories 1, 2, and 6, but if it is only numeric, you may specify 7, 8, or 9. TIDE doesn't care about letters vs. numbers, but when reading in the data later, it may be simpler for you to be able to declare the variable an integer as opposed to character (string) variable. Since TIDE makes up an SPSS syntax command to read in the data, it needs to know which of these is the case. **Note:** TIDE does not check to see whether your codebook incorrectly allows for non-number choices for types 7-9; this will only create a problem when you try to use TIDE's syntax file to read in the data later.[*]

[*] If you want to be very tricky, you may be able to get away with declaring response categories that are numeric, but having your categories be Y and N (to simplify entry for a yes/no question), and then do a global search and replace on the data file Y to 1 and N to 2, adapt the SPS value labels as well, and read in the data as numeric.

Right after the character is the value label that tells the operator what this means.
IMPORTANT: TIDE does **not** allow $ for a character input for a single value input form (e.g. 1, 2, or 6 above). This is used to indicate to TIDE that you are passing a system command in place of data.

If the question is a "range" question, then instead we may have a high and a low delimiter. For question 6, {L}0 says that answers cannot be less than 0, and {H}24 says they can't be higher than 24. If you don't set a range, TIDE assumes it's a four digit integer. If you need more than four digits plus or minus, or five digits plus (such as would be the case if you were entering a phone number), the program may need to be modified.

Response category 15 allows you to give an indeterminate number of response categories that TIDE will number; this may be useful when you want your values to be all numbers for ease of using with other programs (e.g. SPSS), but you have more than 10 and so too many for the single digit entry. It is also useful for when you are still adding response categories but beware! Data entered before response categories have been added cannot be read in with SPS files created after categories have been added, **unless you have added these categories at the end**. Here's why.

To do this, instead of giving a number or letter for each response category, you just write a #. TIDE will sequentially number these for its interface with you and SPSS (though they remain un-numbered in the codebook file). So here is an example:

```
{{Q#}}Author Affiliation
{END15}
{#}Public University with Graduate Department
{#}Private Univeristy with Graduate Department
{#}State College without Graduate Department
{#SQ+2}Private  College without Graduate Department
{#SQ+2}Community College
{#SQ+2}Think-tank
{#SQ+2}Government
{#}Independent Scholar
{#}Federal Institute (NIMH, NIDA, NSF, NIA)
{#}Other
{#}Foreign university
{{NEXT}}
```

(The skips are put in here just to demonstrate that everything else remains the same.) It will appear to you as

```
Q#   1
Author Affiliation

 Valid Response categories
 ( 1)   Public University with Graduate Department
 ( 2)   Private Univeristy with Graduate Department
 ( 3)   State College without Graduate Department
 ( 4)   Private  College without Graduate Department
 ( 5)   Community College
 ( 6)   Think-tank
 ( 7)   Government
 ( 8)   Independent Scholar
 ( 9)   Federal Institute (NIMH, NIDA, NSF, NIA)
 (10)   Other
 (11)   Foreign university
```

Note that it is important for such responses that you not give leading blanks in your answer (i.e. don't press the space bar key before your number). Since you may only go up to 99, TIDE expects only two digits. If you realized you wanted to add another category, "consultant," and you put it right after "Independent Scholar," you would find that the number of Federal Institute had changed to 10, and so on. Hence once data collection as started, you should add new categories at the bottom.

! TIDE does not currently except non-integer numbers as input. If this is required, the program must be modified.

After the response categories or range markers, we see another word in double brackets. {{NEXT}} means we're on to the next question, and {{FIN}} means we're at the end of the codebook. It's basically that simple. We now go on to discuss more complicated structural elements.

B. Skip Patterns

The simplest structural complication is to skip over some questions. There are two types of skips, conditional and unconditional. **Conditional skips only work for choice data** (and not range data). In a conditional skip, you put the question to follow next with the variable label choice, right after the value, by inserting SQ and then the question number to which TIDE should SKIP. This reference can be either **absolute** or **relative**. Thus in the example below, those who have not been married will skip to question 7, and not be asked whether they have ever been divorced. As you'll see in section VII, when you set up a codebook, you won't actually refer to numbers in their four digit (absolute) form for skips, but only in the relative form in terms of a displacement from the current position. However, TIDE converts that to absolute form, which is presented here to make clear the structure of the codebook.

An unconditional skip goes after a question is complete, in place of a {{NEXT}} command. You really should never have to use unconditional skips; a thoughtful design before the fact should allow a better flow. But if you've got two chunks alternating, you can have them proceed in vaguely parallel order through an unconditional skip, which is of the form {{SKIPQ####}} where #### is the four digit question number.

Note: You can only skip forward in a questionnaire, and not backwards. If the question number that is indicated in the skip is not present, TIDE will get to the end, inform you, and abort. Also note that you **cannot** skip from a choice of "other" that takes TIDE to free response, i.e. a & response (see above section A). Have all the other responses skip somewhere. If you use skips, you may end up with multiple {{FIN}}s. This is acceptable. **You cannot skip out of a loop**. You may use multiple $NOLOOP commands. See below on combining SKIPs and LOOPs or IFs.

```
{{Q0005}}Have-you-ever-been-married?
{END2}
{1}Yes
```

```
{2SQ0007}No
{{NEXT}}
{{Q0006}}Have-you-ever-been-divorced?
{END1}
{1}Yes
{2}No
{{NEXT}}
{{Q0007}}Have-you-ever-had-any-children?
{END1}
{1}Yes
{2}No
{{SKIPQ0012}}
{{Q0008}}Etc..
```

C. Pre-Entry Codes

There are a very few "pre-entry" codes that you may use to set general operating parameters for TIDE; these should come at the very beginning of the program (the beginning of the "DECLARATION"s section, as explained below. These are preceded with a double $. The first two pertain to whether TIDE automatically asks for a name (first and last) for each case. This is not written to the file, but is helpful in linking log files to case numbers. The default for this option is "on." To turn this off, put $$NOASKNAME at the top of the file. To turn it on (though this isn't necessary, you should probably include it if it is important, as future versions of TIDE may have a different default), just put $$ASKNAME.

The second pertains to whether TIDE gives an automatic case number to every record. The default for this is no, in which case you are prompted to supply a case or ID number. If you don't want to supply one , TIDE will assign a number (1 greater than the last number it finds in the data file). To turn this on, put $$AUTONUM, to turn it off, $$NOAUTONUM (the default).

III. OPERATION

A. General

Operation of TIDE is pretty straightforward—you do what it says. TIDE prints out most things for the interviewer to read out; but at some times it needs to ask a question of the interviewer him-/herself. In this case, the message to the interviewer is in parentheses. Also, TIDE refers to the interviewer as the "operator," since TIDE may be used for data entry purposes. In this documentation, however, we use the word interviewer to avoid confusing this person with TIDE's OPERators.

It is strongly recommended that if TIDE is being used to enter data in CATI fashion, you have a paper version of the questionnaire handy, in case some error or power shortage leaves you in the lurch. Remember, before beginning you need an ID number for the person interviewed, and any data files that TIDE will need to access in their proper form (see section V below).

B. Files in and Out

When you start up TIDE, you need to know the name of the codebook. If, for example, the codebook is called SURVEY1.CBK you can just tell TIDE to look for SURVEY1. If it can't find it, it will tell you (and don't worry if it calls it a "data" file). You also can choose the files for TIDE to write the data to; by default, TIDE will write the main data to a file with the same name as your codebook, but with an extension .DAT. So for this case, it would write the data to SURVEY1.DAT. If there already is an existing file SURVEY1.DAT, TIDE will add whatever new data comes in to this file; it will also back up the old version under the name SURVEY1.BAK. (This is in case a catastrophic error corrupts the file while data entry is taking place.) TIDE saves the data periodically, but it is always possible for a catastrophic hardware error to lose it, so make sure that you back up this file regularly. If there already is a SURVEY1.BAK, this is deleted.

Until you complete your entry for the person currently in question, his or her data has not been added to the main data file. However, three special backup files are updated with each additional question. One with a .BK1 extension to the base name has the main data, another with a .BK2 extension has dyadic (NAME LOOP) data, and a third with a .BK3 extension has internal TIDE LOOP data. These data are in the form of a list with each person's id number, the question number, and then the response (for the loop data, the object number comes after the person's id number.)For any respondents(s), it updates two files, both with the same basic name but different extensions. One, with a .dat extension, takes most of the data. The other, with a .oth extension, takes "other" data; all data that follows a free-form. See section (III) for how the data is arranged. If TIDE ever encounters any difficulty in reading the codebook, it will immediately abort.

C. Routine Operation

You first give the full name and id number of the respondent. THIS IS IMPORTANT, otherwise you may forget who said what. The full name is not written to the data file, only to the log file; if there is later a confusion about the id number, you can search through the log files and see what name is associated with the ID number. So make sure people have an ID number first.

TIDE will immediately start going through the codebook, telling you the questions and printing out the response categories. If you try to select one that isn't acceptable given the codebook, TIDE will stop you, make an ugly noise, and reprint the valid categories. There is a (limited) ability to go back and correct mistakes, described in the next section, but always have a pen and questionnaire ready to mark changes you may need to make later.

As noted before, for open questions, you start typing whatever the person is saying, but **don't** let the answer run past the line; instead, press return and start a new line. Then, when you are really done, have the last line say END. So, for example, here is input in response to the question, "what was the most important experience of your life?"

```
>I think the most important experience was probably having first child,
>partly because it turned me into a mother but also because until then
>I had not really felt like there was anything I could contribute to
>this world.
>END
```

Note: TIDE is case-insensitive for this END command; to keep from getting confused in the case in which you happen to have a line beginning with "end" (for example: "the most important thing in my life was / ending my marriage…."), TIDE only interprets END as an instruction to finish if there are only three characters entered in the line.

Again, as noted, for some closed choice questions, it may be possible to get to an open response. If this is possible, TIDE will inform you by appending to the choices a note saying to enter & for an extended "other" response. If you type one of these, you then do the same thing as the open question.

When you reach the end, you are asked if you want to go do another person. If yes, we start again, if no, TIDE ends…after manipulating the data somewhat.

D. Real Time Changes and Fixes

If all progresses as planned, TIDE will walk through the questionnaire in a logical fashion. However, if you need to suddenly skip ahead to a different question in a way that was **not** planned for, you may enter $SKIPQ<Number> for any question's data. This passes the message to TIDE to skip ahead to that number. Note that TIDE will not read text preceding this question.

You will recall that you cannot skip out of a loop. To exit a loop, give the $POP command. This will bring you to the first question out of the loop. Then you may skip further. To abort the rest of a loop for one object, and to go on to the next, give the $CYCLE command. Thus if you are asking a lot of questions about 10 group members, and the respondent doesn't remember the fourth, giving the $CYCLE response will skip to the fifth.

Further, you cannot POP, SKIP, or CYCLE in the middle of a sorting task (response type 12). You can, however, ESCAPE, which simply aborts the sort.

- Finally, you may anticipate that a respondent will refuse to answer a whole block of questions, and you wish to comply without aborting the interview. At any place in the codebook, you may place a {{MARK}} (that's it, the whole line). Normally, {{MARK}} will be ignored. But during execution, if you enter $ESCAPE in response to some question, TIDE will SKIP to the MARK. If there is no MARK, TIDE will end with an error message. You may have multiple MARKs, and you may ESCAPE multiple times. **Note**: It may be possible to ESCAPE out of a LOOP, but it is preferable first to POP out of the LOOP and then to ESCAPE. ESCAPE is especially dangerous, because it can be placed anywhere, where the person who designed the codebook can't build in a safety. You might skip out of a set of IF-ENDIF blocks which would throw off all later execution. So TIDE will warn you if you are in a structure before you escape, and you should place MARKs so as to minimize the temptation to jump in an dout of structures.

When you SKIP or ESCAPE past part of the codebook, commands will be ignored, just as if they were placed in an IF block that does not execute. If some of these commands manipulate information in a way you will need it, you cannot skip past them. Put them somewhere where you will not be tempted during execution to bypass them.

- If you make a mistake entering data, you have a limited ability to go back and correct it. To do this, you type $BACKUP at the next available time (when asked for the next question's data). If BACKUP is allowed (more soon), you will return to the previous question and then to the question where you were. BACKUP is **not** always allowed, for the following reason: you cannot BACKUP into or out of a loop structure. That means that if the question you made a mistake on was the last question in a loop, or the last question before a loop, you cannot correct it using BACKUP. Similarly, if you made a mistake regarding data for one object, you cannot change it if you are entering data on a new object. TIDE should keep you from doing this by simply saying that BACKUP is not currently allowed. You cannot backup during LIST sorting events, or during multiple punch questions, which TIDE thinks of as LISTs.

! Furthermore, you cannot successfully salvage a false branch with BACKUP. That is, if question 2 is Yes/No, with a 'No' response leading to a skip to question 5, if the interviewer types in 'No' (when the correct answer is 'Yes'), TIDE will skip to Q5. If you then BACKUP, you can change the answer to question 2, but TIDE will then return to question 5.

IV. LISTS

If any such complicated designs are necessary, your codebook (which is actually a program) will have a section of DECLARATIONS that precedes the regular text. A declaration is a way of bringing something to TIDE's attention that it will need to know later. The goal of this section is accordingly to DECLARE to TIDE what it will need to know. The order of these statements is important, since most declarations refer to something that must previously have been declared; if they are out of order, TIDE will be unable to continue. While it is possible to postpone some declarations until they are needed, it is better to have all the declarations at the beginning; this way, if anything is amiss, TIDE will find out before you are in the middle of an interview.

All declaration statements and control execution statements begin with a $; in particular, DECLARations bgin with $DECLARE:. There are four types of things you DECLARE to TIDE:

1) FILEs that it needs to know about;
2) RESPonse CATegories (which are discussed in section VII)
3) Principles of execution, such as whether files are grouped, whether network data is considered reflexive or not
4) And variables that will control execution
5) LISTS of objects.

Here we describe the last of these.

As explained in greater detail in the next section, there are a number of circumstances in which a respondent is asked to evaluate a set of other "objects." In some cases, these lead to dyadic data structures, in which the same questions are asked of multiple objects. In these cases, the implementations discussed in the next section are likely to be used. In other cases, one single question is asked of each object, as is the case when objects are to be ranked. In this case, it is more parsimonious to tell TIDE that there is a LIST of objects, and to use a particular question to rank them. This allows each object to have a particular place in the data structure (so that we know what the rank of object 1 is for all respondents, as opposed to knowing what any respondent ranked first). This would be used in conjunction with response type 12, for example.

Note that LISTs are assumed to be no greater than 100 elements; hence the rank of any LIST element in a response type 12 question is a three digit integer. Each element in a list is only 60 characters long at most.

To give TIDE a LIST, you simply DECLARE it in the introductory section as follows: $DECLARE:LIST#<list number>. Then each list item follows on a separate line, and finally $ENDDECLARE. When you are ready to use a LIST, right before the question that involves the list, give the USE command with the number of the LIST to be used. Here is a valid example.

```
$DECLARE:LIST#1
Coca-Cola
Pepsi
Sprite
Fanta
RC Cola
Yoo-Hoo
$ENDDECLARE
….
$USE(1)
{{Q#}}Please rank the following soft drinks in how bad you believe they are
for your overall health, with the worst one first.
{END12}
{{NEXT}
```

You **must** precede every LIST-using question with a USE command; otherwise, the question-numbering routine that TIDE uses to get up a codebook will not recognize the need to insert additional question numbers. So even if you have two LIST-using questions, one after the other, using the same LIST, you still need a USE command in front of each. Similarly, a USE command that does not directly precede a LIST-using question will also lead to a crash.

A. <u>Sorting</u>
As you can see, this usage is a sorting task. During execution, TIDE will first direct the interviewer to tell us whether the respondent needs help in ranking the items, or whether they are pre-ranked. If they are pre-ranked, the respondent simply reads off the elements in ranked order; the interviewer then types in what the respondent says, and TIDE looks it up on the LIST.

! TIDE only searches for the first ten characters to make a match; and even if fewer characters serve to uniquely identify an item, that will be sufficient. However, LIST items cannot have the first ten letters in common, or TIDE will not be able to distinguish them. If it is necessary to have two identical phrases, simply insert a non-readable character before one, and instruct your interviewer to enter this but not to read it. Thus if you had “Macintosh II” and “Macintosh III”, you might change the first to “*Macintosh II.”

When you enter a list item, you do not need to type the whole line. Thus for the above example, you could enter “Yoo” to indicate “Yoo-Hoo.” Here TIDE ignores upper- vs. lower-case differences.

➢ If your respondent does not remember all the items, just type $LIST when asked for the next one. TIDE will print out all the items on the LIST that have not yet been picked.

If the respondent does need assistance, TIDE operates somewhat differently. The respondent will first be asked to rank each item low, medium, or high, and then to assign a rank within that category. The items will appear in the data file as separate variable numbers; consequently, the next question number will be <u>six</u> places away from the question number that appears in the finished codebook. **Take this into account if you are skipping over a list question**. So in your codebook, this question might be turned to Q0012; in your data file, you would see questions 12, 13, 14, 15, 16, 17 corresponding to each of the six sodas. The next question (whatever it is) would be Q0018. If you want to skip over this loop, from, say, Q0011, you would SKIP +7, not +2.

If you skip into the middle of a LIST, TIDE will crash, because these questions do not exist in the codebook. This would occur if from Q0011 you tried to skip +4.

B. <u>Multiple Punch</u>

For some questions, you may want to give respondents the option of selecting more than one option, which we refer (in accordance with custom) as a "multiple punch." You would handle this with a LIST construct, which would be expanded to a set of questions, similar to the previous example. This uses and END 13 category. Here is an example:

```
$DECLARE:LIST#2
Happy
Depressed
Excited
Anxious
Peaceful
$ENDDECLARE
….
$USE(2)
{{Q#}}Tell me, since you woke up this morning, whether you felt any of the
following?
{END13}
{{NEXT}
```

If this was the 8th question, the next one in the codebook would be 13. In your data file, question 8 would be "HAPPY," question 9 "DEPRESSED," etc. 1 indicates that the respondent selected this item, and 0 that he did not. The operator simply types in all the words chosen; with the proviso identical to that in the previous section that only the first 10 characters are consulted for a match. If there is a confusion, the whole set of responses must be indicated. These all go on a single line.

- If you want to indicate that the respondent has chosen all possibilities, put EVERY as the response. These will be scored '5' as opposed to '1' to indicate that the choices came in the form of "everything" or "everyone.

V. USE OF OPERATORS

A. General

In some cases, you may wish TIDE to alter its questions, formats, etc., on the basis of information pertaining to some individual. In this case, you DECLARE to TIDE that a FILE will be used, and transfer information to what TIDE calls an operator (a variable referred to as an OPER that is used externally to control program flow). Four types of operators are currently supported: (1) external or USER, (2) internal or TIDE, (3) RANDOM and (4) file-based. These are described in order.

(1) An external or USER operator is supplied by the user at the time of running. It is only different from a "variable"—that is, the information you give in response to a question asked of the respondent—in that it is not written to the file, and simplifies a set of jumps. To declare a USER operator, you simply write $DECLARE:OPER#1@USER. When TIDE encounters this, you will be asked to supply the value for this operator. Note that a colon follows DECLARE; the number of the operator follows a cross hatch; this is then followed by an 'at' sign, and the name of the source of the operator then follows (USER). **Note**: Because it gets hard to keep track of all the OPERators, you may refer to each by a mnemonic term, as discussed in section VIIIB.

(2) An internal or TIDE operator is one that is set on the basis of data received. It allows you to have multiple branches depending on one piece of information. To do this, you need to first declare the TIDE operator, and then **immediately after** the new data has been collected, you $ASSIGN:OPER#<number>. This data **must** be of an integer form. It is acceptable for the data to be entered in a choice format (e.g. 1, 2, or 6), but if the entry has been made to be some non-number (such as 'Y'), TIDE will assign the operator 0 and notify you of an error. After this value has been assigned to the operator, the operator can be used in IF statements.

! If you have a conditional SKIP in some question, don't use the ASSIGN command, since you may skip over this value being ASSIGNed. You should probably first use ASSIGN, and then IF structures as opposed to a conditional SKIP.

(3) A RANDOM operator is to allow for experimental designs. If you are creating a RANDOM variable, you simply declare it by giving it a unique number, and a lower and an upper bound. (All random variables are integers.) Thus

```
$DECLARE:OPER#2@RANDOM[1,5]
```

tells TIDE that OPERator 2 is a random variable from 1 to 5 inclusive. Note that a colon follows DECLARE; the number of the operator follows a cross hatch; this is then followed by an 'at' sign, and the name of the source of the operator then follows, along with brackets containing the bounds.

(4) If TIDE will need to consult an external data file to construct an operator, you must **begin** the codebook by declaring these files. The general format for a declaration is:

$DECLARE:<Type of file>@'<Name of file>',<# variables>

Note that "type of file" refers to a four-letter term. Currently there are four: NAME, DYAD, GRUP (short for Group level) and INDI (short for individual-level). No spaces are present. For example, here we declare a dyadic file.

```
$DECLARE:DYAD@'relation.dat',3
```

The next section has more information on declaring files; here we assume that files are declared and concentrate on getting information from files into operators.

B. Using Operators

The idea of using files is to take information from them and to use it as a TIDE OPERator, which is also done by declaration. Here, you simply set an operator to the ith variable in some declared file. Thus the syntax is

```
$DECLARE:OPER#4@INDI[3]
```

to tell TIDE to set the fourth operator to the third variable in the INDI file (not including the id number which is always the first variable). If there is any missing data, remember to code it to an integer value; otherwise TIDE may be utterly confused at trying to read in a blank.

You may have up to 99 operators; operators do not need to be contiguous; thus you may define operators 1, 3, 5, and 10. On the use of operators to control program flow, see section III.

You may move information from a "variable" (coming from the interview) to an operator with the ASSIGN command discussed above; you may also go the other way, and turn an OPERator into a variable with the STORE command. For example, you may wish to save the value of an operator that is created during execution to the data file. This would be the case for RANDOM or TIDE operators, which do not come from files. Most importantly, in an experimental situation, you may want to record which treatment the respondent received. To do this, you first set up your operator, which should have a value, and then set up a dummy question in the codebook. The question has an END code of 0; there may be a question text given and response categories listed. These are not given to the respondent, not printed at the time of the interview, but you may want to include them so that they will be written to the SPSS syntax file. Immediately after you list this question (and before a {{NEXT}} command, you tell TIDE to STORE some operator (given by its number); TIDE will write this to the data file for this question. Here is an example:

```
$DECLARE:OPER#3@RANDOM[1,2]
...
{{NEXT}}
{{Q#}}Random Allocation of Direction of Wording
{0}Positive
{1}Negative
{END0}
$STORE:OPER#3
{{NEXT}}
```

Note that such data must be 4 digit integers; character data is not permitted. If TIDE does not encounter a STORE command, it ignores the question and writes a warning. Finally, you may put an integer into an operator with the POKE command. The syntax here is $POKE:OPER#<operator number>(<value>). You could use this to determine whether you are in the first iteration of a LOOP or not, for example, to avoid redundant text.

C. Use of the Stack

In most cases, it is easier to do any manipulations in the files that you will read in, as opposed to doing them within TIDE. But it is possible to do any logical or integer manipulation using OPERators through ASSIGNs, IFs, POKEs, and STOREs. To make these less cumbersome, basic arithmetic manipulations can be carried out using a stack. A stack is a LIFO (last-in, first-out) ordering of numbers; "manipulations" are carried out on the last numbers in the stack. You can PUSH an OPERator onto the stack, carry out manipulations, and then PULL the answer off the stack into an OPERator (which should be defined as TIDE).[*] With stack-based arithmetic, to add 4 and 4, you would first push 4, then push 5, then add. 4*(3+2) would be Push 3, push 2, add, push 4, multiply.

TIDE allows you to $ADD, subtract ($MINUS) and $MULTiply with the stack. You can also change a negative number to a positive, or a positive to a negative with $FLIP (which is equivalent to pushing –1 and $MULTiplying). It is sometimes easier to use multiplication of 1/0 variables to construct new operators for IF structures than to have multiple IFs. Here is an example of a structure to ask people who lost their job if they have had stress, but only if they have not answered a previous questionnaire and are a parent.

```
$DECLARE:INDI@'olddata.dat',3
$DECLARE:OPER#1@INDI[2]
! oper 1 is a dummy that is 1 if we have recent data on this person
$DECLARE:OPER#2@TIDE
$DECLARE:OPER#3@TIDE
{{Q#}}Do you have any children?
{END9}
{0}No
{1}Yes
{{NEXT}}
$ASSIGN:OPER#2
$PUSH:OPER#2
{{Q#}}Have you been fired from a job in the last six months?
{END9}
{0}No
{1}Yes
{{NEXT}}
$ASSIGN:OPER#2
$PUSH:OPER#2
$PUSH:OPER#1
$MULT
$MULT
$PULL:OPER#3
$IF(OPER#3)=1
  {{Q#}}Have you recently felt like you had more stress than you could take?
  {END9}
  {0}No
```

[*] It is possible to PULL information to a non-TIDE operator, but this is bad practice.

```
  {1}Yes
  {{NEXT}}
$ENDIF
…
```

If you are subtracting, the second value on the stack is subtracted from the first; thus PUSH 5, PUSH 3, MINUS leaves 2 on the stack. Many other uses may be found for these manipulations, but it is better to have as many manipulations be done in external files.

Other Stack Commands:
You may take the absolute value of whatever is on top of the stack with $ABS. You may transform a non-zero number to zero, and zero to a 1, with $NOT. Note that $NOT $NOT binarizes what is on the number.

You may $DUPlicate the top value on the stack with $DUP. To eliminate the whole stack, use $PURGE.

Note: An operation by default removes from the stack any elements that it uses. If you want to preserve them, use the $DUP command. Here is what happens with the above example, "4*(3+2)", assuming a 1 was already on the stack:

Push 3	Push 2	Add	Push 4	Multiply	
	2		4		(top of stack)
3	3	5	5	20	
1	1	1	1	1	(bottom of stack)

- To throw away something from the stack, just pull it to an OPERator and write over it or ignore it.

With judicious use of the stack, pretty much anything can be accomplished. Here, for example, is (obsolescent) code that made up for the fact that IF statements did not originally use less than (<). We want to write $IF(OPER#1<1964)….

```
$POKE:OPER#2(1964)
$PUSH:OPER#2
$PUSH:OPER#1
$DUP
$MINUS
$ABS
$MINUS
$PULL:OPER#3
$IF(OPER#3=0)
```

(If OPER#1<1964; X=1964-OPER#1>0; hence X-X=0.)

D. Debugging

When you use many operators to control program execution (see section on conditions, section VI below), you may find that a codebook executes differently from your intent. To figure out what is wrong, you can ask TIDE to tell you the current value of all operators at any point with the command $SPIT. This information will be written to the log file, and not the screen. If you want to keep track of operators and the stack as they change, use the $DEBUG command. After this command, most changes will be written to the log file. To stop this writing, use the $NODEBUG command.

VI. USE OF FILES AND LOOPS

A. <u>Types of Files</u>

1. Right now, you may only declare one file at each of these levels; also note that group level files are not necessary, since it is possible to make an individual level file that contains the same information; group files are simply more parsimonious.

2. Note that in addition to any data you read in, TIDE demands that the first variable in the INDI file be the person's id number (the same TIDE will use); the first variable in the GRUP file is the group number; the first two variables in the DYAD file are the two corresponding to ego (the respondent) and alter; in that order. **DO NOT** count these when you compute the number of variables to be read in. Finally, note that this must be in the directory that TIDE will be operating in, or else you must specify a path for the file.

3. For NAME files, you do not need to specify the number of variables, it is always two. Names files are special files that contain the names of things or persons which will be used to construct items in real time. A NAME file first has the number of names to follow, then each line has an ID number for the name (the same as person IDs if these are dyadic data referring to other persons in some group), and then the name. This name must be **45 Characters or Less**.

4. If you are using DYADic data that refers to numbers including the interviewee's ID number, you can declare that reflexive data not be asked (which is the default), or that it can be asked. To declare reflexive data as valid, say:
   `$DECLARE:REFLEXIVE`
   To declare reflexive data as invalid, say:
   `$DECLARE:IRREFLEXIVE`

5. The order in which the files are declared is arbitrary with the following exception: if you wish to use separate group-files at the dyadic level or for names files, (see below, section on group Files), you need to declare the individual level file first thing.

6. Finally, TIDE is currently set up so that you may have only 95 variables or names.

If you mistakenly tell TIDE to read in more variables for some file than are actually written, TIDE will "loop over" and read in the next data line, leading to it only reading in every other case. It may then crash if it can't find needed data. If you get an error that it can't find a match for the current case in the data file, and you are sure that this case is there, see if you have misspecified the number of variables.

B. <u>Grouped Files</u>

In many cases, people may belong to different groups, and you wish to construct DYADic data from the NAMEs of group members. In this case, you will clearly need a NAME file for each group separately. Similarly, you may have a DYAD file for each group separately. If this is the case, declare that your data files are grouped by putting $DECLARE:GROUPS(V#) right after you declare your individual file. You need an individual level file, since TIDE needs it to know what group any person is in. The V# is the number of the variable in the INDIvidual level file (not counting the first) which contains the group number for each person.

If you are going to use these multiple group files, you must have less than 100 different groups, and their numbers must be positive integers<100. You still declare file names for NAME and DYAD files, but you declare basic name type, which will be transformed into a file for each group. The rule for these name types is that the **second** and **third** characters are used to hold the group numbers. Thus g01dyad.dat is a valid name. There must be a leading zero for numbers less than 10. So you would

```
$DECLARE:DYAD@'g##dyad.dat',3
```

(the ## are just as place savers).

During operation, if you say you are entering information for person 34, TIDE will open your individual level file you have declared, go to the variable you claim has information about group membership, and retrieve this. Imagine that this is group 3. It will then insert '03' into the filename you supplied, and look to open `'g03dyad.dat'`. If this is not present, TIDE will necessarily abort.

REMEMBER: FIRST you must declare your individual file with $DECLARE:INDI, which must be global, not group specific. SECOND you must declare that the other files are grouped, with $DECLARE:GROUPS(), specifying the group number variable. THIRD you must declare NAME or DYAD files.

C. Use of Dyadic Files

For some questions, we ask each respondent about a set of other stimuli, which TIDE considers "objects." It is assumed that these objects are most likely to be other people, and hence these data will generally be assumed to be dyadic. The NAMEs data is assumed to have NAMEs of persons, but they could be NAMEs of brands of soap. Similarly, it is possible for "DYADic" data to link persons to objects. Note that if objects do not refer to persons, TIDE will still assume they do if the ID numbers of objects are the same as the ID numbers of persons. So make sure to DECLARE your data REFLEXIVE if DYADic data structures are used to refer to non-person objects.

For the purposes of this documentation, it will generally be assumed that NAME and DYADic files refer to persons as the objects. In such a DYADIC structure, since the number of questions asked will depend on the number of objects, there is no fixed number of questions. This does not affect the numbering of the main questionnaire, since a separate dyadic level file is made with fixed questions. To enter dyadic data, you declare the NAME file, and program in a LOOP as discussed in the next section. Whether questions in this loop are executed may depend on some dyadic level variables; if this is the case (and it need not be) then you must have declared a dyadic level file.

If you have declared the data to be IRREFLEXIVE (see above), if TIDE comes to a name with the same number as the current respondent, it will **not** ask about this name. If you have declared REFLEXIVE, it **will** ask about this name.

D. Getting information from Files

There may be some circumstances in which you want to get information from the files, but not through the normal DECLARation of an OPERator linked to a file. This is because TIDE links the OPERator to the FILE using characteristics about the current respondent. You might, however, wish to access information about some other respondent. To do this, you use the $PEEK command. The syntax for PEEK is $PEEK:<filetype>[<variable number>] where <filetype> is DYAD, INDI or GRUP. You cannot currently PEEK into a NAME file. How does TIDE know which record to find? You have previously PUSHed this information onto the stack. (If you are accessing DYADic data, you would first push the alter-id number, and then the respondent id number.) TIDE will remove this information from the stack, and leave the requested value. Here is an example (typically heteronormal in its assumptions, but what the heck) in which we are interested in getting the sex of alter for each person in a NAME LOOP, and using this for a conditional branch. As explained in the next section, $PUSH:ALTER is used to get alter's id number on the stack.

```
$DECLARE:MNEME@'EgoSex'
$DECLARE:MNEME@'AlterSex'
$DECLARE:MNEME@'Samesex'
$DECALRE:MNEME@'scratch'
$DECLARE:Egosex@INDI[3]
$DECLARE:Samesex@TIDE
$DECLARE:scratch@TIDE
…
$LOOP:NAME
  $POKE:samesex(0)
  $PUSH:ALTER
  $PEEK:INDI(3)
  $DUP
  $PULL:AlterSex
  $PUSH:Egosex
  $MINUS
  $PULL:Scratch
  $IF(Scratch=0)
    $POKE:Samesex(1)
  $END IF
  $IF(samesex=1)
    {{Q#}}Do you consider [NAME] a role model?
    RESPCAT#2
  $ELSE
    {{Q#}} Do you find [NAME] attractive?
    RESPCAT#2
  $END IF
$NOLOOP
```

Note that alter's sex is saved even though it is not used in this example.

E. <u>Use of Loops</u>

LOOPs allow you to ask the same questions in modified form. There are three types of LOOPs—generic FILE loops, NAME loops, and Internal (TIDE) loops. A loop begins with a $LOOP command as a single line, contains a block of text and questions, and ends with $NOLOOP. Any LOOP goes "over" some variable that may take on multiple values. These are referred to here as "objects." Because of this, the number of items is not fixed before hand (but see (2) below). Consequently, these data are saved in a different file from the main data, and questions are numbered independently of the main questions.

1) FILE implementation
   LOOP will eventually allow for loops over any file you choose, for any variable in that file, but now it is only implemented for the NAME file.

2) NAMEs implementation
   This implementation is slightly different from the other LOOP implementations, in that we know in advance how many questions will be asked, and hence can compose a dyadic data structure corresponding to a rectangular matrix. These data are hence written to a separate file, with the filename <base>.DYD, where <base> is the codebook name or other specified name for the data files. Here we have a list of names in the names file, and we wish to ask each respondent the same question about each of these names (or some of them, depending on dyadic data). The questions that pertain to these names all are placed within a single loop.

   When LOOPing over names, simply put $LOOP:NAME. Each time you progress through a loop, another name has been selected from the NAME file. At any point in the question or in TALK, you may make reference to this name simply by writing [NAME]. (Note that this enclosure between square brackets is a general way of alerting TIDE that a text variable needs to be replaced at each iteration of a LOOP.) This [NAME] will be replaced as TIDE reaches it with the current name. The same name can be used in multiple questions. You may also need at some points to refer to the ID number of this person (see above, section D). The simple command $PUSH:ALTER puts alter's ID number on the stack. (If given outside of a NAME LOOP this results in an error.) $PUSH:ID leaves the respondents id on the stack.

3) Internal (TIDE) Implementation
   In other cases, you may not know what you will need to loop over; that's what you're asking about. For example, you may wish to ask people about each of their children, and they may have had from 0 to 10 children. You might write 11 sets of items, with a skip pattern to take you across all inapplicable ones, or you can use an internal loop. Here you begin with $LOOP:TIDE. Make sure your codebook skips over this whole loop if it is not applicable. Every question in the block enclosed by $LOOP and $NOLOOP will be asked, and repeated until you POP out of the LOOP. To do this, simply have your last question be "do you have another X on which to report?" with No leading to a POP (see below on POPs and CYCLEs).

All values will be written to a special file for such internal looped data; this has the basename that the main data file has, but an extension of .LOO. In this file, first there will be the respondent's ID#, then the number of the object (LOOP number), then the number of the question, and then the data. You may write questions so as to insert the LOOP number, just as you can write questions to insert the [NAME]. LOOP numbers may be inserted as ordinal or cardinal numbers; for the former, put [ORD] in your text, and for the latter, put [CARD]. ORDinal numbers are written in words up to 10, and then as numbers later. Here is an example of an Internal LOOP. Note that the filtering question simply says "skip to the next question," but TIDE recognizes that questions within the LOOP are not "next" for purposes of skipping. This could also be done with an Internal Operator and an IF structure, as introduced in the next section.

```
{{Q#}}Do you have any children?
{END1}
{1}Yes
{2SQ+1}No
{{NEXT}}
{{TALK}}Think of all of your children, from the oldest to the youngest.
{{NEXT}}
$LOOP:TIDE
  {{TALK}}Let me ask you about your [ORD] child.
  {{Q#}}When was this child born?
  {END4}
  {L}1920
  {H}2000
  {{NEXT}}
  {{Q#}}Is this child a boy or a girl?
  {END2}
  {1}Boy
  {2}Girl
  {{NEXT}}
  {{Q#}}Do you have any more children?
  {END1}
  {1}Yes
  {2POP}No
  {{NEXT}}
$NOLOOP
{{Q#}}Do you ever….
```

Note how indentations are used to highlight the structural organization; while indents cannot be part of the final TIDE codebook, they can be used in the form that TIDE will read. But all indentation must be done with the space key; **do not** use tabs.

CYCLES and POPs

If you want to have a conditional structure at the last point in some LOOP, you may use a CYCLE branch. Instead of writing {<value>SQ+<skip number>} for some value in your list of response categories, you substitute CYCLE for SQ+<skip number>. When TIDE hits this, it will immediately restart the LOOP at the top, eliminating following questions. Here is a valid example.

```
$LOOP:NAME
  {{TALK}}Let me ask you about [NAME].
  {{NEXT}}
  {{Q#}}Around how often do you see [NAME]?
  {END1}
  {1CYCLE}At least once a month
  {2CYCLE}A few times a year
  {3}Maybe once a year
  {4}At least once in the past 10 years
  {5}At least once since we stopped living together
  {6}I have not seen this person since we lived together
  {7CYCLE}I don't even remember this person.
  {{NEXT}}
  {{Q#}}Would you like to see [NAME] more often?
  {1}Yes
  {2}No
$NOLOOP
```

You may also use a POP command in place of a CYCLE; POP, however, totally terminates the loop process, and does not ask about further objects. This might be incorporated if it is believed that respondents are likely to refuse to answer similar items. Such POPs (and CYCLEs) can also be entered by the operator as real time commands (see below).

CAUTIONS:

1) See below on the combination of loops and conditional (IF) structures
2) You may skip within loops, but you may not skip out of loops nor into loops. If you attempt to do this, TIDE will shut down. You may skip **over** a LOOP. TIDE will ignore questions within a LOOP when figuring out the questions numbers, so you may use relative displacements that refer only to the non-LOOP questions. If, by any chance, you have two identical loops in different places (e.g. two NAME loops), and you want to skip from a question in one to a question in the other, you may do so by (1) declaring a TIDE OPERator; (2) ASSIGNning that OPERator some value within the first loop; (3) POPping out of the first loop; (4) uing an IF statement with the OPERator to jump over to the second loop; (5) entering that loop; (6) using an IF statement with the same OPERator to jump over some questions within that LOOP.

VII. USE OF CONDITIONS

The reason to read in the external files is to allow TIDE to skip questions depending on some variable value. The syntax for such a command is:
$IF(OPER#<operator number><relation><value>)
…
By relation, we mean =, <, or >; by value, we mean an **integer** value. String and real numbers cannot currently be used for conditions. This integer must be <1,000,000,000.

An operator can be considered strictly Boolean, in that it only has two values, zero and non-zero (some stack manipulations can convert integer into Boolean values). For the sake of simplicity, if you treat an operator as Boolean, you may simply write
$IF(OPER#<operator number>)
The following code will execute if the operator is non-zero.

There are two basic conditional structures: (1) IF/END IF; (2) IF/ELSE/ENDIF.
(1) In an IF/END IF structure, anything between the $IF and $ENDIF lines is excuted only if the specified condition is true. Otherwise, execution resumes after $ENDIF.
(2) In an IF/ELSE/ENDIF, $ELSE divides the code into two sections. The first portion, between the $IF and the $ELSE, is executed if the statement is true. The second portion, between $ELSE and $ENDIF, is executed if the statement is false.

You may also use a set of “ELSE IF”, but these behave differently from a standard ELSE IF in that the options are not exclusive; this is because TIDE converts ELSE IF to an END IF and an IF statement. So if you were to write
```
$IF(OPER#3=1)
  POKE:OPER#2(1)
$ELSEIF(OPER#3=2)
  POKE:OPER#3(1)
$ELSEIF(OPER#3=4)
  POKE:OPER#4(1)
$END IF
```

This would be converted to
```
$IF(OPER#3=1)
  POKE:OPER#2(1)
$END IF
$IF(OPER#3=2)
  POKE:OPER#3(1)
$END IF
#IF(OPER#3=4)
  POKE:OPER#4(1)
$END IF
```

In this example, the unpacked version behaves as a normal ELSEIF construct, because a single operator can only have one value. But in the next example, because two operators are used, this is not so. (MNEMonics are used here to illustrate the logic of the example, although we haven’t gotten to them yet.)

You type in

```
$POKE:famstatus(0)
$IF(remarried)
  POKE:famstatus(3)
$ELSEIF(divorced)
  POKE:famstatus(2)
$ELSEIF(married)
  POKE:famstatus(1)
$END IF
```

The logic here is that you want to make a single variable that tells whether the respondent is never married (0), has married and is currently married (1), has gotten divorced but not remarried (2), has divorced and remarried (3). This logic would normally work, since you start with the "rarest" thing (remarriage), and if that isn't true, pass on to the more "common" status (by definition, since remarriage means a marriage and a divorce).

But since TIDE treats this as

```
$POKE:famstatus(0)
$IF(remarried)
  POKE:famstatus(3)
$END IF
$IF(divorced)
  POKE:famstatus(2)
$END IF
$IF(married)
  POKE:famstatus(1)
$END IF
```

You will find that all your divorced and remarried people end up having their values reset to 1 (since they have gotten married). In sum, ELSEIF works find when your conditions use a single variable, which is the more common case (you want to split up some answer to make a set of dummies to control program execution); if you are going to use more than one variable in conditions, you will need to think carefully and most likely reverse the order you would normally choose, if not abandon the ELSEIF construct altogether. It is really just there to make the code more readable.

Some other important cautions:
These if structures are the most powerful part of a system like TIDE, and consequently great care has to be taken in their use.

1) You must be very careful not to have and $IF without an $ENDIF, or vice versa, and the same goes for $ELSE.
2) **!** Note that **nested IFs are possible**. But they are tricky, and because they lead to multiple branches, make sure you always pre-test every possible combination before using on real data.
3) It is your responsibility to make sure you are not SKIPping across blocks. It is possible to SKIP out of an IF block, although this is discouraged. TIDE should be able to keep track of where you were in terms of nested IFs and cancel as many as necessary. But if you try to SKIP **into** an IF block, the condition for which the IF was evaluated will never have been tested.

4) Note that if $IFs are used to divide blocks containing LOOP data questions, the question numbers must be non-overlapping (as opposed to having multiple ways of asking the same question embedded in different loops). This is good practice for all data (though it is conceivable to use an $IF construct to divide question text while having the data go to the same question number): you can always combine things later. If TIDE does the numbering for you, it should correctly number the LOOP questions consecutively. Below is an example of such an IF/ELSE/ENDIF struture embedding alternate loops. Note that the loop command occurs within the IF structure blocks, which is good practice. (Also note that while question numbers have been changed to absolute reference, as if TIDE had edited the codebook, leading blanks have been left in to highlight structure.)

```
$IF(OPER#3=1)
!  Respondent has familiarity with names.
  $LOOP:NAME
    {{TALK}}These questions are about [NAME].
    {{NEXT}}
    {{Q0001}}I approve of how this person has handled the job
    {END1}
    {1}Yes
    {2}No
    {{NEXT}}
    {{Q0002}}I would vote for [NAME] again.
    {END1}
    {1}Yes
    {2}No
    {{NEXT}}
  $NOLOOP
$ELSE
!  Here the respondent lacks familiarity.
  $LOOP:NAME
    {{Q0003}}Have you by any chance heard of [NAME]?
    {END1}
    {1}Yes
    {2}No
    {{NEXT}}
  $NOLOOP
$ENDIF
{{TALK}}Now we would like to ask some questions about family life…
```

VIII. HAVING TIDE SET UP THE CODEBOOK FOR YOU

A. Bases: Question numbers and skips

Setting up a TIDE-ready codebook really isn't as hard as it looks, but clearly, it makes inserting a question difficult—you need to renumber everything afterwards! For this reason, you should set up your codebook without question numbers, and let TIDE fill them in when it reads in your codebook the first time. It will then make a copy of your old codebook with the base name and the extension ".OLD", and save the finished form. Here's what you do.

A. When printing standard questions, instead of putting Q and the four digit item number in curly brackets, just put {{Q#}}. Exactly like that—no extra spaces, no extra #'s.

B. When referring to skip patterns, make them relative to the current position. This is basically the same for conditional and unconditional skips. For an unconditional skip, instead of writing {{SKIPQ0012}} (as in the example above), you just write {{SKIPQ+5}} (since we skipped from the 7th to the 12th question). NOTE: This isn't the number of items you skip over, it's the number ahead to go to. SKIPQ+1 is just where you'd be going anyway. Since you're going to want to do this via addition and subtraction in your head anyway, we might as well formalize it. So you just count until you get to where you want to be. Similarly, for a conditional skip (to use the one from question 5 above), you would write instead of {2SQ0007} {2SQ+2}.

C. **NOTE**: When using TIDE to adjust skips, you may not skip more than 98 questions down. That is, '+99' is the largest skip. If this is a problem, the program must be adjusted.

If you are having TIDE create the final codebook (**strongly recommended**), just start up TIDE the first time and supply the codebook you have created. TIDE will be able to tell that it needs to edit the codebook. This only occurs **once**—the new codebook is saved, and until you change it, you don't need to re-fix it. Also, remember to save the .OLD file—that's what you make changes to. If you lose it, there's no advantage to ever having made it. TIDE also saves a file with the extension '.COL' that it needs to use to create an SPSS syntax file. If this gets destroyed, re-create the codebook by changing your OLD file to the CBK extension.

NOTE that if you use loops in your program, TIDE will give these question numbers their own numbering system, and if there is more than one loop, the numbering in the second loop will pick up where those in the first loop left off. TIDE will also put a line $DONE at the top of the codebook to tell it not to try to renumber it. NAMEs LOOPs and TIDE LOOPs have independent numbering systems, since they go to independent files.

B. Response Categories and Mnemonics

It may also be the case that you use the same response categories over and over again. You can cut and paste in your editor, but you can also have TIDE do this as it sets up a codebook. To do this, you declare a set of response categories with a unique number; later, you simply refer to that number. When TIDE makes a codebook, it will make these insertions and remove your statement of declaration. You may have up to 99 such declarations. The syntax is (for declaring the nth set of response categories) $DECLARE:RESPCAT#n; you then write the {END} statement that includes information about the category types, all the categories, include the {{NEXT}} statement if you wish, and close with #ENDDECLARE. Here is an example.

```
$DECLARE:RESPCAT#1
  {END6}
  {1} Yes
  {2} No
  {3} Maybe
  {{NEXT}}
$ENDDECLARE
```

When you wish to refer to this category, after a question, do **not** place an END statement, and instead simply write $RESPCAT#1. The text between the $DECLARE and $ENDDECLARE statements will be repeated verbatim.

Note: If you do this, and you happen to use a RESPCAT for the LAST question in the codebook, TIDE will insert a {{NEXT}} before the {{FIN}}, when actually, there is not {{NEXT}} for the last question (see example). If you do this, TIDE will return an error, so remember to change the codebook by hand before running, or don't use a RESPCAT for the last question.

Similarly, you may use mnemonics to refer to OPERators. To do this, you must declare a MNEME **before** you make reference to the operator. When TIDE sets up the codebook, it will assign this MNEME an operator number, and transform every reference in a command line **or** a comment line to the MNEME to a reference to this operator. The syntax for declaring a MNEME is simply $DECLARE:MNEME@'<name>'. The MNEME name **must** be under 12 characters. If it exceeds 12 characters, it will be truncated. MNEME names are case insensitive. Here is the original code for an example given in the section on operators that has been written using MNEMEs.

```
$DECLARE:INDI@'olddata.dat',3
$DECLARE:MNEME@'NewData'
$DECLARE:MNEME@'work'
$DECLARE:MNEME@'result'
$DECLARE:NewData@INDI[2]
! NewData is a dummy that is 1 if we have recent data on this person
$DECLARE:WORK@TIDE
$DECLARE:result@TIDE
{{Q#}}Do you have any children?
{END9}
{0}No
{1}Yes
{{NEXT}}
$ASSIGN:WORK
$PUSH:WORK
{{Q#}}Have you been fired from a job in the last six months?
{END9}
{0}No
{1}Yes
{{NEXT}}
$ASSIGN:work
$PUSH:work
```

```
$PUSH:newdata
$MULT
$MULT
$PULL:Result
$IF(Result=1)
  {{Q#}}Have you recently felt like you had more stress than you could take?
  {END9}
  {0}No
  {1}Yes
  {{NEXT}}
$ENDIF
```

As with response categories, you may have only 100 MNEMEs.

Note: You really should avoid mixing up MNEMEs and straight-out operator declarations, to avoid the possibility of you and TIDE using the same operator in different ways. However, it is possible to do this; just count up the number of MNEMEs you have, and make sure any OPERators you refer to by number start about the number of MNEMEs.

Note: TIDE will replace MNEMEs with OPERator numbers in your comment lines to assist you in following the program flow through the final codebook. For this reason, avoid having MNEMEs be words that you will use in other contexts in your comments. That is, if you call a MNEME 'gender' and then have a comment line, '! Now we ask about gender roles' you will find your comment line has become '! Now we ask about OPER#4 roles.'

➢ To have TIDE tell you what each operator number means, simply have a comment line saying the operator name in its MNEME form, and what it refers to. TIDE will replace this with its number.

! TIDE replaces MNEMEs in the order they appear. If the text of one MNEME includes that of a second, put the former (the longer one) first. For example, if you want two MNEMEs, 'done' and 'redone,' if you declare 'done' first, it will be set to OPER#1 (say), and everywhere where 'redone' appears will be set to 'reOPER#1.' 'redone' will never be found. So DECLARE 'redone' first, and 'done' second.

C. Formatting Issues.

Lines cannot exceed 90 or so characters. If a line is too long, TIDE will abort, and notify you of the oversized line. But you should be aware that the length of any line can grow, and to plan accordingly. First of all, when TIDE changes # to question numbers, it adds spaces, but when there is a substitution of a [NAME] or an [ORD]inal number, the line also grows. **Plan accordingly!**

The actual codebook TIDE uses is assumed not to have blank spaces at the beginning of lines. But the codebook you make may have such blanks; TIDE will remove them as it adjusts the numbering. These blanks may be used to indent nested blocks in LOOPs or IF statements.
Note: While you may insert blanks at the beginning of lines, do **not** use TABs. Depending on your editor, these may stay as strange characters that will confuse TIDE. Just use the spacebar.

You may have a blank line in the codebook written for TIDE to change, but TIDE will not copy it.

All command and structure words require that they be spelled exactly as shown, without blanks inserted. However, if you have TIDE set up the codebook for you, you may have blanks inserted in **command lines** only which it will remove (this is because structure commands may contain text). So you may write $END IF instead of $ENDIF. But this is not so if TIDE is **not** setting up the codebook for you.

Note: When going setting up the codebook, TIDE will catch some errors, but it does not do a complete syntax check. Most errors will remain until you actually try to enter data. Therefore you must test out your program well before actually using it.

IX. READING IN THE DATA

A. General Issues

Because TIDE allows for character and non-character input of at least four different types, reading in the final data is not necessarily so simple. However, TIDE makes this easy by writing an SPSS syntax file that will read in all the data that it has written, with the exception of the "OTHer" data file for in-depth responses. This file has the same root name as the codebook or data file, but with an '.SPS' extension. It distinguishes between numeric and non-numeric data, and writes value labels from the codebook to the file. It also gives variable labels, that are simply the first 60 characters of the question wording. You should open this file for minor editing. First of all, TIDE doesn't put the name of the data files in quotation marks, since it doesn't know what directory you'll end up putting the data file in. So you supply the exact name. Second, TIDE will include a variable name definition for type 5 (long answer) data that is not actually present in the file. Take these out if you don't want to see the error messages from SPSS.

Note: These files are created every time TIDE sets up a codebook from scratch. If you erase these files by mistake, you will need to rename your OLD file CBK and restart TIDE.

If you have internal (TIDE) LOOP data or external (NAME) LOOP data, TIDE will have written separate files for these data; the syntax for reading in these data will be in the same file as the main syntax.

To create this data file, TIDE of course needs to know what columns it will be writing the data to. If TIDE has just set up a codebook, it will have all this information, and will write it to a file called <codebook name>.COL. If you have already set up your codebook, TIDE will look for this file. If it does not exist, TIDE will abort, and you will need to reset up your codebook (unless you can find the file). **Note**: If you have made changes to a codebook by hand, TIDE will not have changed the column information in the .COL file, and so the new data will appended to old data with the wrong columns. This is one reason to always let TIDE make changes in codebooks.

Note: Currently, if you split up loops, so that you loop over the same people twice, TIDE will write multiple records for these files, which will be blank in some places. If you read in the data, simply aggregate afterwards over PERSID and ALTER or OBJECT number.

B. Creating Variable Names

TIDE automatically names each variable its number, which is extremely handy if you are going to set up a codebook, leave it alone, and then collect all your data. But a change made to the codebook in the middle of data collection will produce two files which diverge in the meaning of the names past some point. (That is, if you insert a question 8, Q0008 no longer refers to the same item when you read in the data; nor does Q0009, Q0010, etc.) This can be handled by a lot of renamings, but it can also be handled by labeling your questions in the codebook.

To do this, before you come to your question, you tell TIDE to use a label. This is then applied to the next question read as a name. If there is no recent unused label, TIDE gives its default name based on the number. Here is an example.

```
$LABEL:age
{{Q}}What was your age on your last birthday?
{END3}
{L}10
{H}90
{{NEXT}}
$LABEL:religion
{{Q}}What best describes your religious background?
{END7}
{1}Protestant
{2}Catholic
{3}Jewish
{4}Muslim
{5}Buddhist
{6}Hindu
{7}Jain
{8}Taoist/Confucianist
{9}Pagan
{{NEXT}}
```

This produces the following syntax file excerpts:

```
DATA LIST  FILE=test1.DAT
  RECORDS=1
/1
 RespID 001-005
 age 006-010
religion 011-011 .
VARIABLE LABELS
 RespID 'Respondent ID Number'
  age ' What was your age on your last birthday? '
religion ' What best describes your religious background? '     .
VALUE LABELS
/religion
1 'Protestant'  2 'Catholic'  3 'Jewish'  4 'Muslim'  5 'Buddhist'  6 'Hindu'
7 'Jain'  8 'Taoist/Confucianist'  9 'Pagan'
.
```

Note: TIDE allows this name to be 12 letters, while SPSS requires only 8 letters. You are responsible for your labels being the right length. (TIDE is more generous anticipating a change in SPSS.)

X. SYNTAX CHART

In this chart, key commands and words are given in bold—examples in plain type, explanations in italics.

Codebook Structure

{{Q#}}	*Tells TIDE that the text of a question follows*
{{END#}}	*Tells TIDE that the text of the question is over, and what type of answer to expect*
{{FIN}}	*Tells TIDE the interview or coding is over*
{{LINE}}	*Tells TIDE to print a blank line*
{{MARK}}	*Establishes a place for the operator to ESCAPE to*
{{PAUSE}}	*Prompts operator for CR before continuing*
{{NEXT}}	*Tells TIDE we are on to the next question or command*
{{TALK}}	*Tells TIDE this is text to be read*

Values

Types

Skips

{<value>SQ<number or +displacement>}<value label>

CYCLEs

{<value>CYCLE}<value label>

POPs

{<value>POP}<value label>

Pre-Entry Commands

$$ASKNAME	*Turns on TIDE's asking for subject's name*
$$AUTONUM	*Turns on TIDE's assigning each case an automatic case number*
$$NOASKNAME	*Turns off TIDE's asking for subject's name*
$$NOAUTONUM	*Turns off TIDE's assigning each case an automatic case number*

Codebook Commands

$ABS *Takes the absolute value of the topmost stack element and places it on the stack*

$ADD

$ASSIGN

$ASSIGN:OPER#<operator number>. Assigns the most recent value to this operator; is 0 if this last value was not an integer.

$CYCLE *Skips to next iteration of a LOOP*

$DECLARE

FILES <DYAD/INDI/GRUP>

$DECLARE:<filetype>@'<filename>',<number of variables>

GROUPS *Used to tell TIDE that there are multiple groups with separate files*

$DECLARE:GROUPS(< variable number containing group information>)

LIST *Establishes a list to be used by type 12 and 13 responses*
$DECLARE:LIST#< number of this list>
<element 1>
<element 2>
…
<element N>
$ENDDECLARE

MNEME
$DECLARE:MNEME@'<name>'

OPERator

RANDOM
$DECLARE:OPER#<number>@RANDOM[<low>,<high>]
USER *Tells TIDE that this operator will be set at run-time by user*
$DECLARE:OPER#<number>@USER
TIDE *Tells TIDE that this operator will be set by respondent's answer*
$DECLARE:OPER#<number>@TIDE
<file>
$DECLARE:OPER#<number>@<filetype>[<variable number>]

IRREFLEXIVE
$DECLARE:IRREFLEXIVE

REFLEXIVE
$DECLARE:REFLEXIVE

RESPCAT *Used to contain information on response categories that will be used for multiple questions*
$DECLARE:RESPCAT#<response category number>
<<END statement, response categories, NEXT statement>>
$ENDDECLARE

$DEBUG	*Turns on internal debugging; any changes to operators noted in logfile*
$DUP	*Takes the topmost stack element and places a duplicate on the stack*
$ELSE	(see $IF)
$ELSEIF	(see $IF); *This is included to be compatible with programming conventions, but it is converted to $END IF; $IF… in the codebook used by TIDE.*
$ENDDECLARE	(see $DECLARE:RESPCAT)
$ENDIF	(see $IF)

$IF
$IF(OPER#<operator number><relation: =, <, or > <value>)
or
$IF(OPER#<operator number>) *for Boolean operators*

<<codebook commands>>
$ELSEIF(OPER#<operator number><relation:=, <, or > <value>) (optional)
<<codebook commands>>
$ELSE (optional)
<<codebook commands>>
$ENDIF

$FLIP	*Changes negative to positive and positive to negative for top number on stack*
$LABEL $LABEL:<label>	*Gives a character label that will be used when writing SPSS syntax file; this will become the variable name.*

$LOOP

GENERAL
$LOOP:<filetype>,<Variable number>
<<codebook commands>>
$NOLOOP

INTERNAL
$LOOP:TIDE

NAMES
$LOOP:NAME
<<codebook commands>>
$NOLOOP

$MINUS	
$MULT	
$NODEBUG	*Turns off internal debugging. See* $DEBUG
$NOLOOP	(see $LOOP)
$NOT	*Logical negation of top element on stack*
$PEEK<file> $PEEK<filetype>[<variable number>]	*Gets information about some other person from a file*

$POP	*Exits a LOOP structure*
$POKE	*Put an integer value directly into an OPERator*

$POKE:OPER#<operator number>(<value>)

$PULL *Pulls a value from the stack to an OPERator*
$PULL:OPER#<operator number>

$PURGE	*Eliminates all values on stack*
$PUSH	*Pushes some value onto the stack*
OPERator	*Pushes a value from an OPERator to the stack*

$PUSH:OPER#<operator number>

Respondent ID
$PUSH:ID
Alter's ID
$PUSH:ALTER

$SPIT *Writes operator values and stack contents to logfile*

$STORE *Turns an OPERator into a variable written to the data file*
$STORE:OPER#<operator number>

$USE *Tells TIDE that the next question refers to the following LIST structure*
$USE(<listnumber>)

Real Time Commands

$BACKUP	
$CYCLE	
$ESCAPE	*Jumps ahead to next section of questionnaire as given by {{MARK}}*
$LIST	*Prints unchosen items in a LIST during ranking or choosing operation*
$POP	
$SKIP	

<filetype>=INDI, GRUP or DYAD